线性表及其在民航系统中的应用实验报告

姓名：王琪博　　学号：2023200091

2024 年 10 月 7 日

1　问题分析

本实验首先要求将数据存储在结构体数组中，并按照一定的规则进行排序和查找。第二部分是将指定的结构体组成链表，并对该链表进行排序和查找。

2　数据结构设计与实现

2.1　ADT 设计

- 数据对象：航班数据构成的结构体
- 数据关系：线性关系
- 基本操作：数组初始化，增添，修改，删除，查询
- 数据对象：航班数据构成的结构体
- 数据关系：线性关系
- 基本操作：链表初始化，增添，修改，删除节点，删除链表，查询

2.2　ADT 的物理实现

需要列出构建数据对象的代码片段。

- C 语言代码展示举例，C++ 语言只需修改参数为 `[language=C++]`

```
typedef struct planeInfo{
    int flightID;
    int departureDate;
    std::string intlOrDome;
    int flightNo;
    int departureAirportID;
    int arrivalAirportID;
    double departureTime;
    double arrivalTime;
    int airplaneID;
```

```
    int airplaneModel;
    int airFares;
    int peakSeasonRates;
    int offSeasonRates;
    struct planeInfo* next;
}planeInfo, * planeNode;

std::string filename = "D:\\data.csv";
flight myFlight;  // 创建flight对象实例
// 构建结构体数组
std::vector<flight::planeInfo>* flightData = myFlight.
   readFlightInformation(filename);

// 构建结构体链表
flight::planeNode flightSystem::newFlightList(std::vector<flight::
   planeInfo>* flightData, int planeId) {
    std::vector<flight::planeInfo> onePlaneData;

    for (const auto& plane: *flightData) {
        if (planeId == plane.airplaneID) {
            onePlaneData.push_back(plane);
        }
    }

    std::sort(onePlaneData.begin(), onePlaneData.end(), compare);

    flight::planeNode head = new flight::planeInfo;
    flight::planeNode current;
    current = head;

    for (const auto& plane : onePlaneData) {
        flight::planeNode newNode = new flight::planeInfo;
        *newNode = plane;
        newNode->next = nullptr;

        current->next = newNode;
        current = newNode;
    }
    return head;
}
```

3 算法设计与实现

3.1 读写文件算法

读写时主要借助 fstream 函数，逐行读入 csv，然后将每一行根据相应的数据类型进行解析读入结构体中

3.1.1 算法分析

- 时间复杂度分析：对于包含 n 行数据的文件，程序需要对每一行进行读取和处理，因此读取操作的时间复杂度为 O(n)。每行数据通过 `parseCsvLine` 函数解析，该函数对固定数量的字段进行处理，时间复杂度为 O(1)。因此总体的时间复杂度为 O(n)。
- 空间复杂度分析：数据存储部分：需要存储 n 个 `planeInfo` 对象，空间复杂度为 O(n)。临时变量部分：使用了字符串和字符串流等临时变量，其空间与单行数据的大小有关，属于常数级别，空间复杂度为 O(1)。总体空间复杂综合以上，程序的空间复杂度为 O(n)。

3.1.2 算法实现

Listing 1: parseCsvLine

```
    flight::planeInfo parseCsvLine(const std::string& line) {
    std::stringstream lineStream(line);
    std::string cell;
    flight::planeInfo emp;

    std::getline(lineStream, cell, ',');
    emp.flightID = std::stoi(cell);  // int

    std::getline(lineStream, cell, ',');
    emp.intlOrDome = cell;  // std::string
    //以下同理

    return emp;
}
```

parseCsv 函数负责将读取到 csv 文件的每一行进行解析，将不同数据类型的数据读入结构体

Listing 2: read_csv

```
std::vector<flight::planeInfo>* flight::readFlightInformation(std::
    string fileName) {

    std::ifstream flightFile(fileName, std::ios::in);
```

```
    std::string line;
    auto emp = new std::vector<planeInfo>;
    getline(flightFile, line);
    while (getline(flightFile, line)) {
        planeInfo emp1 = parseCsvLine(line);
        emp->push_back(emp1);
    }
    return emp;
 // 省略篇幅节省部分代码片段
}
```

这一个函数主要使用 ifstream 将将每一行 csv 读取出来，解析后读入结构体

3.2 排序文件算法

实验中总共有两次排序操作，排序的约束和条件都相同，不同的是一个是数组排序，另一个是链表排序。因此可以将链表的数据重新写入数组，这样可以用相同的函数解决问题。排序算法主要依靠 C++ 标准库中的 sort 函数，底层实现是快速排序。

3.2.1 算法分析

- 时间复杂度分析：快速排序的时间复杂度是 O(nlogn)，将链表读入数组的时间复杂度是 O(n)，因此最终的时间复杂度是 O(nlogn)。
- 空间复杂度分析：需要存储 n 个 `planeInfo` 对象，空间复杂度为 O(n)。第二次排序借助外部数组，空间复杂度为 O(n)，最后的空间复杂度为 O(n)。

3.2.2 算法实现

```
    std::sort(flightData->begin(), flightData->end(), cmp);

    bool cmp(flight::planeInfo a, flight::planeInfo b) {
    if (a.departureAirportID != b.departureAirportID)  return a.
       departureAirportID > b.departureAirportID;
    else {
        if (a.arrivalAirportID != b.arrivalAirportID) {
            return a.arrivalAirportID > b.arrivalAirportID;
        }
        else {
            return (a.arrivalTime - a.departureTime) < (b.arrivalTime
                - b.departureTime);
        }
    }
}
```

3.3 查询文件算法

查询算法主要包括三部分，第一部分为查找费用为素数的航班，算法为遍历读入的结构体数组即可。第二部分为查找二维矩阵的鞍点，首先通过遍历得出二维数组，之后根据鞍点的定义遍历二维数组，得到最后的鞍点。第三部分是构建航班组成的链表，首先构筑数组将其按照条件排序，之后将新数组的数据读入结构体。

3.3.1 算法分析

- 时间复杂度分析：查找素数和构建二维数组都是遍历之前的数组，因为时间复杂度都为 $O(n)$; 在查找鞍点的时候，遍历二维数组，时间复杂度为 $O(n^2)$;
- 空间复杂度分析：构建二维数组的空间复杂度为 $O(N*M)$, 其他不占用额外空间，空间复杂度为 $O(1)$。

3.3.2 算法实现

查找素数的函数

```
    // 查找素数
    void flightSystem::findPrimeFareFlight(std::vector<flight::
       planeInfo>* flightData) {
    for (const auto& plane : *flightData) {
        if (flightSystem::isPrime(plane.airFares)) {
            std::cout <<"flight id: "<< plane.flightID << "  basic
               fares: " << plane.airFares << std::endl;
        }
    }
}
```

查找鞍点的函数

```
//初始化二位数组
    td::vector<std::vector<int>> flightSystem::initTwoDimensionVec(
       std::vector<flight::planeInfo>* flightData, int
       departAirportId, int arrivalAirportId);
//判断是否是每行的最大值和每列的最小值
bool
flightSystem::ifRowMaxAndColumnMin(std::vector<std::vector<int>> temp
   , int row, int column) {
    auto maxRowElement = *std::max_element(temp[row].begin()+1, temp[
       row].end());
    int minColumnElement = 100000;
    for (int i = 0; i < temp.size(); i++) {
        if (temp[i][column] < minColumnElement) minColumnElement =
           temp[i][column];
```

```
    }
    if (maxRowElement  == minColumnElement) return true;
    else return false;
}
// 查找鞍点
void flightSystem::findArraySaddlePoints(std::vector<std::vector<int
   >> temp) {
    for (int i = 0; i < temp.size(); i++) {
        for (int j = 1; j < temp[0].size(); j++) {
            if (ifRowMaxAndColumnMin(temp, i, j)) {
                std::cout << "saddle points flight id: " << temp[i
                   ][0] << std::endl;
            }
        }
    }
}
```

构建一个飞机的所有航线的链表，首先构建一个辅助数组，排序完成后再构建链表

```
    flight::planeNode flightSystem::newFlightList(std::vector<flight
       ::planeInfo>* flightData, int planeId) {
    std::vector<flight::planeInfo> onePlaneData;
    for (const auto& plane: *flightData) {
        if (planeId == plane.airplaneID) {
            onePlaneData.push_back(plane);
        }
    }
    std::sort(onePlaneData.begin(), onePlaneData.end(), compare);
    flight::planeNode head = new flight::planeInfo;
    flight::planeNode current;
    current = head;
    for (const auto& plane : onePlaneData) {
        flight::planeNode newNode = new flight::planeInfo;
        *newNode = plane;
        newNode->next = nullptr;

        current->next = newNode;
        current = newNode;
    }
    return head;
}
```

3.4 删除和重组文件算法

第四问的链表构建过程和第三问保持一致，从顺序表中删除的操作可以使用 erase() 函数进行，输出 id 只需要遍历链表就可以完成。上述过程完成后，可以将链表转化为数组，便可以重新实现功能。

3.5 计时算法

3.5.1 算法实现

实验主要使用 chrono.h 头文件，将每一步都嵌入进计时程序中，以便查看每一步的执行时间。

```
auto start = std::chrono::steady_clock::now();
// 嵌入程序
auto finish = std::chrono::steady_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::milliseconds
   >(finish - start);
```

4 实验环境

- 硬件环境：Intel Core Ultra 125H、内存 SK Hynix LPDDR5 32G
- 软件环境：windows 环境，语言为 c++, ide 为 VS2022

5 实验结果与分析

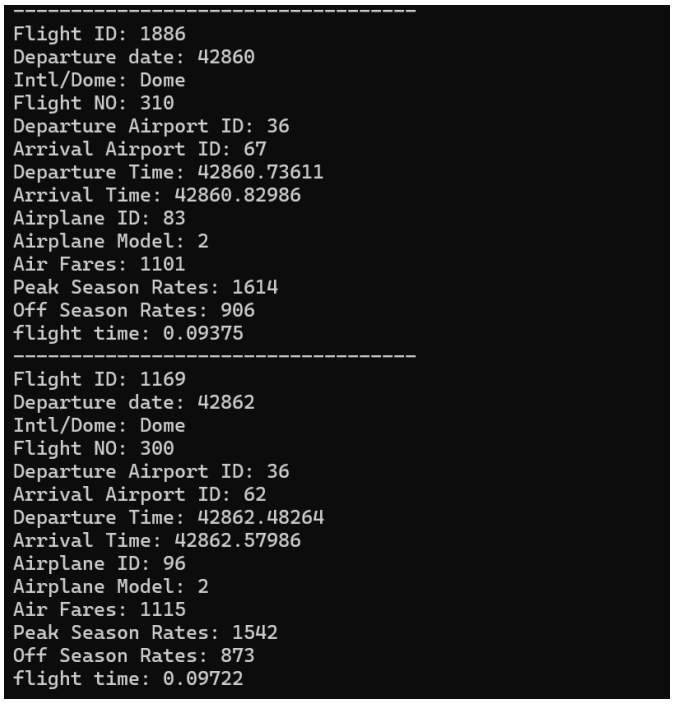

图 1: 排序结果部分展示

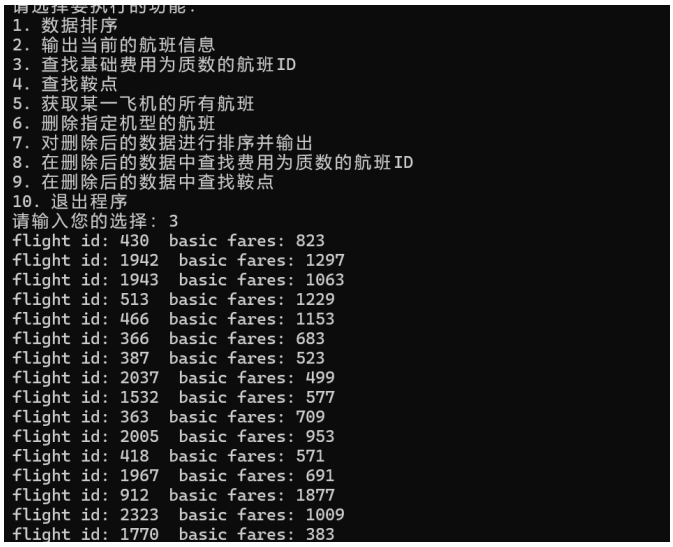

图 2: 查找费用为质数结果部分展示

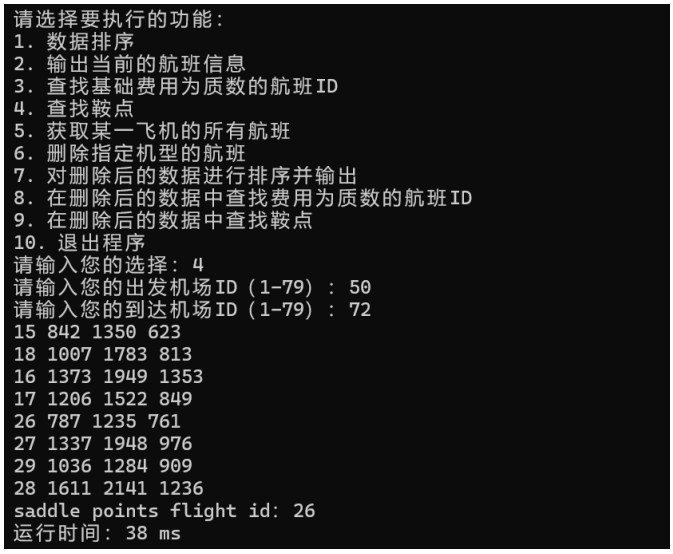

图 3: 鞍点结果展示

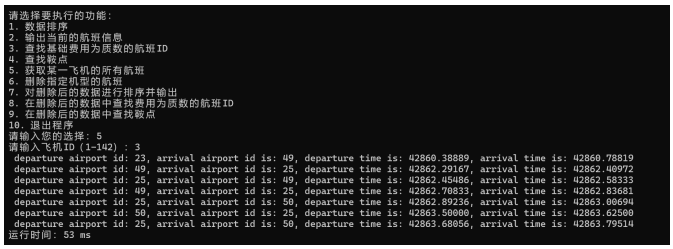

图 4: 查找一架飞机的所有航程结果部分展示

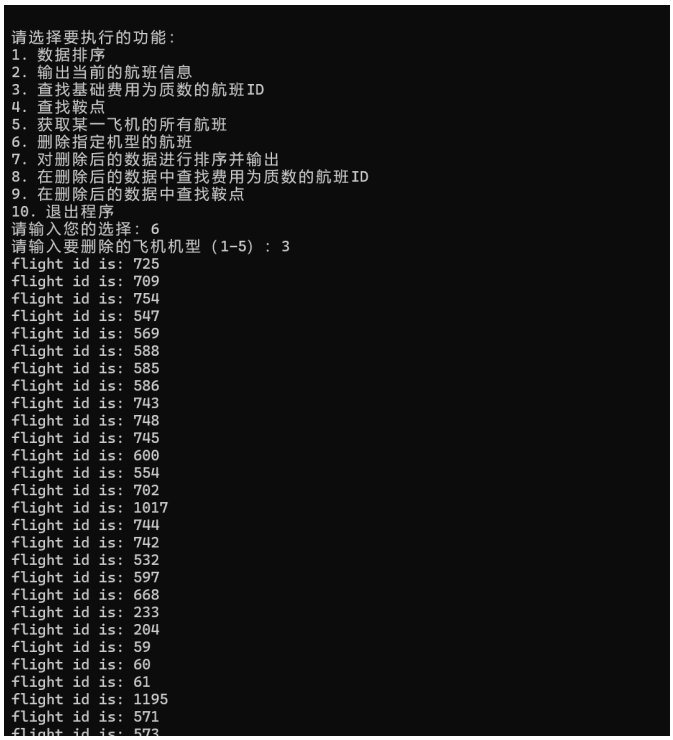

图 5: 删除一个机型的结果展示

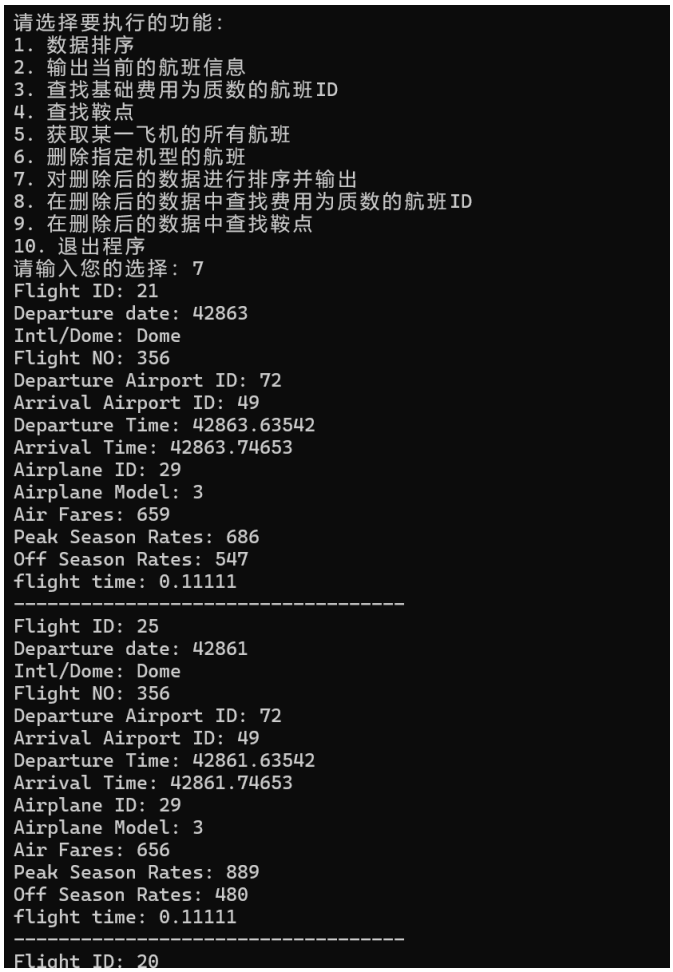

图 6: 删除后的链表的排序

图 7: 删除后查找质数费用的航班

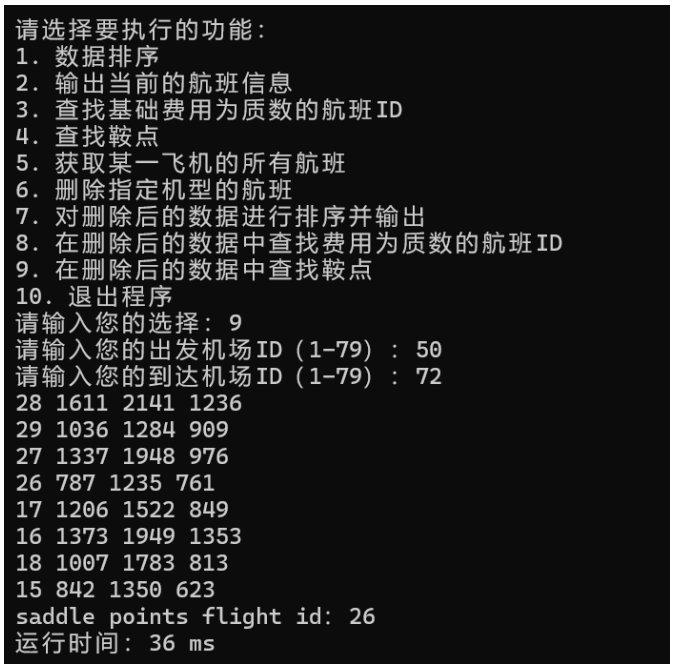

图 8: 删除后的数据查找鞍点